CS-537: Midterm Exam (Spring 2002)

The Mid-Semester Blues: A Tour of Proper Answers

Please Read All Questions Carefully!

There are nine (9) total numbered pages

Please put your name on every page.

Grading Page

	Points	Total Possible
Part I: Short Answers	**60**	$(12 \times 5) \rightarrow 60$
Part II: Long Answers	**40**	$(2 \times 20) \rightarrow 40$
Total	**100**	100

Part I: Short Questions

The following questions require short answers. **Each of 12 is worth 5 points (60 total)**.

1. Processes (or threads) can be in one of three states: **Running**, **Ready**, or **Blocked**. For each of the following four examples, write down which state the process (or thread) is in:

   (a) Waiting in `Domain_Read()` for a message from some other process to arrive.
   Blocked. Waiting for a message in the OS.

   (b) Spin-waiting for a variable `x` to become non-zero.
   Running. By definition, it's spin-waiting, therefore it must be running.

   (c) Having just completed an I/O, waiting to get scheduled again on the CPU.
   Ready. Unblocked but not running means ready.

   (d) Waiting inside of `pthread_cond_wait()` for some other thread to signal it.
   Blocked. Wait gives up the processor, also by definition.

2. An operating system runs in *privileged* mode, a hardware state where it has full access to machine resources. Why is such a mode needed, and why can't normal user processes and threads enter privileged mode?

   Need a privileged mode to allow OS (and only OS) to have full access to machine resources (timer interrupts, MMU, all of physical memory, etc.). If user processes could enter privileged mode, they could do anything they want, including corrupting other processes or even the OS. Protection and multiplexing (basic OS services) could not be reliably provided.

3. In a system with pure paging, assume we have a 32-bit address space, and a 4 KB page size.

   a) How many bits of an address specify the *logical page number* (a.k.a. *the virtual page number*), and how many bits specify the *offset*?

   4KB page implies 12 bits needed for the offset. This leaves 20 bits for the VPN.

   b) Let's say we are translating the logical address 0x00010033; if each logical page is mapped to a physical page that is a single page number higher (i.e., logical page 10 is mapped to physical page 11, logical page 11 is mapped to physical page 12), what is the final translated physical address?

   Have to take VPN and increment it by one. The VPN is the top 20 bits (0x00010). Adding one gets you 0x00011. Combining back with the offset gets you the final answer: 0x00011033.

4. Three jobs (A, B, and C) arrive to the job scheduler at time 0. Job A needs 10 seconds of CPU time, Job B needs 20 seconds, and Job C needs 30 seconds.

   a) What is the *average turnaround time* for the jobs, assuming a *shortest-job-first* (SJF) scheduling policy?

   Turnaround time the total time the job spends in the system (from when it was submitted to when it completes). SJF runs A to completion, then B to completion, then C to completion. A finishes in 10 seconds (A's run time), B in 10 (waiting for A) + 20 (B's run time) = 30 seconds, and C in 10 (A's run time) + 20 (B's run time) + 30 (C's run time) = 60. Average turnaround is thus $\frac{10+30+60}{3}$, or $\frac{100}{3}$ seconds.

   b) What is the *average turnaround time* assuming a *longest-job-first* (LJF) policy?

   Similar reasoning, but in reverse order. C finishes in 30, B in 50, A in 60. Average turnaround is thus $\frac{30+50+60}{3}$, or $\frac{140}{3}$ seconds.

   c) Which finishes first, Job C in SJF or Job A in LJF?

   C finishes last in SJF, and A finishes last in LJF. Thus, neither, as they both finish at the same time.

5. In class, we gave the following code as an implementation of mutual exclusion:

```
boolean lock[0] = lock[1] = false;
int turn = 0;
void deposit (int amount) {
    lock[pid] = true;
    turn = 1 - pid;
    while (lock[1-pid] && (turn == (1 - pid)))
        ; // spin
    balance = balance + amount;
    lock[pid] = false;
}
```

Let's say we replace the statement `turn = 1 - pid` with the statement `turn = BinaryRandom()`, where the function `BinaryRandom()` returns a 1 or 0 at random to whomever calls it. *Will the code still function properly? If so, why, and if not, what problem could occur?*

Does not work. Imagine if two processes enter at roughly the same time. Both could set their respective locks to true, and then set turn equal to `BinaryRandom()`. What if `BinaryRandom()` returns 0 for thread 0 and 1 for thread 1? Mutual exclusion is broken.

6. A number of threads periodically call into the following routine, to make sure that a pipe that is shared between them has already been opened (after calling this routine, a thread might go ahead and call write() on that pipe, for example). Assume there is a global integer `pipe`, which is set to -1 when the pipe is closed, and a global lock `lock`, which is used for synchronization. Here is the code:

```
void MakeSurePipeIsOpen() {
    mutex_lock(&lock);
    if (pipe == -1)
        pipe = open(``/tmp/fifo'', O_WRONLY);
    mutex_unlock(&lock);
}
```

However, you get clever, and decide to re-write the code as follows:

```
void MakeSurePipeIsOpen() {
    if (pipe == -1) {
        mutex_lock(&lock);
        if (pipe == -1)
            pipe = open(``/tmp/fifo'', O_WRONLY);
        mutex_unlock(&lock);
    }
}
```

Does this code still work correctly? If so, what advantage do we gain by using this implementation? If not, why doesn't it work?

Turns out this is OK. Just reading the value of pipe is not a problem, because we are not updating it. What this saves you is the overhead of getting the lock if pipe has already been set to something other than -1.

7. Assume you are implementing a producer-consumer shared buffer (which can be used by producer threads to pass data to consumer threads), but that the buffer is *unbounded*; in other words, it does not have a limit as to how big it can get.

   a) How many condition variables will you need in order to implement this buffer properly, and why?

   This solution only requires one condition variable, in order to wake up consumers when some data is placed into the queue.

   b) How is this different than a standard bounded buffer implementation?

   The traditional bounded buffer requires two condition variables; the additional one is needed to wake producers when the buffer is not full anymore.

8. For deadlock to occur, four conditions must hold: *mutual exclusion*, *hold and wait*, *no preemption*, and *circular wait*. If any one condition does not hold, no deadlock can occur. Assume we want to allow "preemption", and thus get out of deadlocks; in other words, if a deadlock is detected, we will forcibly take a lock away from a thread; by repeatedly doing this, we will eventually undo the deadlock. What new problems are introduced by this preemptive approach?

   Preemption implies that the process that gets preempted to rollback the state changes it has made during the critical section. If you don't do this, preemption simply doesn't work. Rollback is a pain because you must track a lot of extra state in order to be able to do it. Another problem that may be introduced is livelock, depending on how the rollback is implemented.

9. Someone has written new memory allocator to replace the standard malloc()/free() implementation. It works as follows: one half of available memory is divided into fixed-sized units of 4KB, and the other half is managed by a best-fit free list. If an allocation request is less than or equal to 4KB and there is space in the fixed-sized half, a 4KB unit is allocated from the fixed-sized half; otherwise, the best-fit algorithm is used over the other half of memory, and the requested size is returned (if space is available).

   a) Assuming 32KB of total memory is available, what series of allocation requests will most quickly lead to all of memory getting allocated, all while requesting the least total amount of memory?

   Four 1-byte requests and one request for 16KB. The four 1-byte requests each take up one of the 4 4KB pages, and the remaining 16KB is allocated from the best-fit side.

   b) What type(s) of fragmentation occurs with this new allocator?

   Both types. Internal, because we are allocating pages from one-half, and external, because we have a best-fit managed list.

10. A mechanism that can be used for synchronization is the ability to turn on and off interrupts.

a) How can you use this to implement a critical section?

Simple. Turn off interrupts, execute the critical section, turn them back on again.

b) Why does does it work?

It works because you are preventing the timer interrupt (or any interrupt) from letting the OS get control again; when the OS gets control, it could run another job. Without interrupts, the process that is running owns the CPU until the interrupts are enabled or until the processor is voluntarily relinquished.

c) Why is this generally a bad idea?

Bad idea because of *trust*. If you can turn off interrupts, you can keep the processor all to yourself (even by accident). This is not a good idea if you want the OS to arbitrate the resources for you.

11. In class, we talked about two kinds of message sends: *blocking* and *non-blocking*. In communicating through a Unix pipe, consider the sender side (i.e., the side doing the `write()` call to the pipe). Is the `write()` to a pipe blocking, non-blocking, or both? **Explain.**

Both blocking and non-blocking. A write to a pipe is usually non-blocking in that it copies the data into the fifo and then returns, not waiting for anyone to read the data. However, if the pipe is full, the write will block until someone reads from it.

12. For the following question, please **circle all answers that apply.** A translation lookaside buffer (TLB) is generally used to:

(a) translate virtual page numbers into physical page numbers. **yes**

(b) translate physical page numbers into virtual page numbers.

(c) make segmentation have the benefits of a pure paging approach.

(d) translate the addresses generated by loads. **yes**

(e) translate the addresses generated by stores. **yes**

(f) translate the addresses generated by instruction fetches. **yes**

(g) remove the need for a full-sized page table.

(h) make translations happen quickly. **yes**

A TLB translates virtual page numbers into physical page numbers (item a) for addresses generated by loads (d), stores (e), and instruction fetches (f). The idea behind the TLB is to make the process of translation speedy (h).

Part II: Longer Questions

The second half of the exam consists of two longer questions, **each worth 20 points (total 40)**.

1. **Staying In-Bounds.**

   You are dealing with a system that performs *static relocation*. In static relocation, a *loader* rewrites the addresses of a process as it is getting loaded into the system so as to "relocate" the address space of that process to an arbitrary address in physical memory. In this system, **all programs are compiled as if they will get loaded at address 1000**. Then, when the loader is "loading a process", it must re-write any addresses within the program in order to generate addresses at the correct offset in physical memory.

```
load 0(R1), R2   # loads value at address 'R1 + 0' into R2
add R2, 5, R2    # add 5 to R2
store 0(R1), R2  # store value at address 'R1 + 0' back into R2
```

   a): Assuming that the process gets loaded at **physical address 2500**, how would the loader re-write the statements above so as to provide proper static relocation?

```
load 1500(R1), R2   # loads value at address 'R1 + 1500' into R2
add R2, 5, R2       # add 5 to R2
store 1500(R1), R2  # store value at address 'R1 + 1500' back into R2
```

   In this part, all you have to do is to make sure that any address generated by the process is correctly relocated to the right physical address. Of course, only loads and stores (in the example) generate addresses, so they must be rewritten to account for the actual physical placement. In this case, the program was compiled as if it were to get loaded at address 1000, but instead got loaded at address 2500. That means that a load destined for 1000 must somehow get redirected to 2500. The way to accomplish that is to add the difference (2500 minus 1000) to each load and store, which is easily done by using the offset field of the load/store.

   b): Let's say we want to implement some additional checks in our static relocation scheme. Specifically, we want to make sure that all addresses generated by the process do not extend beyond its address space. If an address is outside of the limit, the program should just be forced to exit. What would we have to do before each `load` and `store` instruction in order to guarantee that they stay within the address space of the process?

   We want to make sure that not only do we do the offset (as in part a), but that the offset is legal. Thus, we need to know how big the process address space is (it's bound), and then check each address generated to make sure it is within the bounds of the process (both too high or too low). Thus, before each load or store, one could imagine inserting a few extra instructions to perform the check, and if the check fails (i.e., out of bounds), call exit() to end the process. In a complete solution, one would also need to make sure that control transfers (jumps, branches) were checked.

   c): In contrast with static relocation, **dynamic relocation** is a hardware approach to relocating the address space of a process in physical memory. What **hardware** is required to implement dynamic relocation?

   Not much. Just a base register and a bounds register. (Just saying MMU wasn't specific enough)

   d): If you contrast software-based static relocation with the extra checks (as described in this question in part (b)) to traditional hardware-based dynamic relocation, are they equivalent, or does one approach give you more capabilities than the other? **Explain.**

   Lots of possible answers here. Simple one: hardware is faster, because we don't need all of the extra instructions to do the checks. Also, hardware is better because it allows for us to easily relocate the process when it is getting swapped back into memory. Other answers are possible, i.e., software is better, because it doesn't need hardware support!

2. **Synchronization: Primitive?**

Different hardware architectures provide different low-level instructions to allow one to implement synchronization primitives. In this question, we will examine two different sets of synchronization instructions (available on two different architectures), and will use each of them to implement a critical section.

Load-linked, store-conditional: The first hardware primitive is actually a pair of instructions available on the MIPS architecture, and they are called the *load-linked* and *store-conditional* instructions. They are used in combination to build mutual exclusion.

```
ll <address>, RD
sc <address>, RS
```

In the load-linked instruction (ll), the value at address <address> is placed into the register RD, much like a normal load. With the store-conditional instruction, the value inside of the register RS is placed into the value at <address>, *if the value at <address> has not been changed by some other thread since the load-linked instruction (ll) was executed.* If the store-conditional succeeds (and stores the value in RS into the address <address>), the register RS will be set to the value 1; if the store-conditional fails (in other words, someone else has updated the value at <address> in the meanwhile), the store-conditional does not update the value at <address> and RS is set to the value 0.

Fetch-and-add: The second synchronization instruction is available on the now defunct Alpha architecture, and is called *atomic fetch-and-add* (abbreviated fetchadd). The format of the fetchadd instruction is as follows:

```
fetchadd <address>, RS
```

where <address> holds an address of some variable, and register RS holds an integer value. When fetchadd executes, it atomically adds the value inside of RS to the variable stored at <address>.

The code that must be implemented in properly synchronized form is our standard synchronization routine:

```
int balance = 0; // global variable, accessible by all threads.

void update(int amount) {
    balance = balance + amount; // must synchronize access to 'balance'!
}
```

Your job: implement the update() routine so that it is properly synchronized, using the different synchronization instructions available. In other words, in part a), implement update() by using the load-linked and store-conditional instructions (but not the atomic fetch-and-add or compare-and-swap). In part c), use just the fetch-and-add. Of course, in both parts, you may use other standard instructions such as loads, adds, stores, and so forth.

Assumptions: Assume you have 16 registers at your disposal (you won't need nearly that many), and call them R1 through R16. Also, assume that when update() is called, the value of the amount variable is placed inside of register R1.

You may need to use some other instructions to implement the correct code. To get you started, this is what the unsynchronized version of the update() routine looks like:

```
load <balance>, R2    # load account balance into R2
add R1, R2, R3        # add amount (R1) and balance (R2), result in R3
store <balance>, R3   # store value of R3 back into balance
```

(continued on next page)

You may also need to use a `branch` instruction of some kind. If so, just write some pseudo-C (instead of assembly) and use `goto` statements and labels.

```
top: load <variable>, R1
     if (R1 == 1) goto top
```

In the code snippet above, the variable `variable` keeps getting checked to see if its value has become anything other than 1; as long as it stays at 1, the code keeps branching back to the label `top`.

a): Implement the `update()` routine with the **Load-linked, store-conditional** instructions.

Assume that `amount` **is in R1.**

```
top: ll <balance>, R2          # conditional load of balance
     add R1, R2, R3            # do the add
     sc <balance>, R3          # try to store value into balance
     if (R3 == 0)              # if the store failed, try again!
         goto top;
```

If the store-conditional suceeded, R3 gets the value of 1, and we know that we updated the balance variable atomically. If R3 gets the value of 0, the store-conditional failed, and that means someone else was updating the value of balance at the same time, and did it before we did. Thus, try again!

b): Are there any limitations or problems with your solution to part (a)? If so, please describe them. If not, please say why.

Many possible answers. The solution above spin-waits, which is a waste of processor cycles. Also, livelock is possible (though unlikely), if two threads interleave in such a way as to continually conflict with one another, thus preventing progress from being made.

c): Implement the `update()` routine with the **Atomic fetch-and-add** instruction.

Assume that `amount` **is in R1.**

```
     fetchadd <balance>, R1        # add r1 into balance atomically
```

d): Which of is more appropriate to use in implementing the `update()` routine, the load-linked/store-conditional, or the atomic fetch-and-add? **Explain.**

Fetch-and-add does it all in one instruction, and further, will never fail and require a retry; thus, it avoids spin-waiting and wasting processor cycles. On the other side of the coin, fetch-and-add may be harder to implement in hardware.